Matching in Multi-Agent Pathfinding using M*

Jonathan Dönszelmann[1] , **Jesse Mulderij**[1] , **Mathijs de Weerdt**[1]

[1]TU Delft
jdonszelmann@student.tudelft.nl
{j.mulderij, m.m.deweerdt}@tudelft.nl

Abstract

Multi agent pathfinding (*MAPF*) can be extended by giving agents teams. Agents in a team need to find the best assignment of goals to minimise the *sum of individual costs*. Such an assignment is called a 'matching'. The algorithm *M** is a complete and optimal algorithm to solve *MAPF* without matching. In this paper, a modification was made to *M** to allow it to solve problems which involve matching. To do this, two strategies are proposed called *inmatching* and *prematching*. This paper shows that *prematching* is generally preferable to *inmatching*, it compares the benefits of different optimisations for *M**, and it shows how well *M** stacks up against other *MAPF* solving algorithms extended to perform matching.

Introduction

A large number of real-world situations require the planning of collisionless routes for multiple agents. For example, the routing of trains over a rail network [1], directing robots in warehouses [2] or making sure autonomous cars do not collide on the road [3]. Problems of this nature are called *Multi-agent pathfinding* problems, which in this paper will often be abbreviated to *MAPF*. Solving *MAPF* problems has been proven to be **NP-hard** [4].

One algorithm to solve *MAPF* is called *M** [5]. A standard *A** algorithm as described by Standley [6] plans agents together. This means that in each timestep, the number of possible next states grows exponentially with the number of agents. Contrasting that, in *M**, when possible, agents follow an individually optimal path, and in each timestep, only the subset of agents which is part of a collision is jointly planned.

The *MAPF* problem can be extended by grouping agents into teams. Each team has a number of starts and goals. Within one team, any agent can travel to any goal. In order to solve this problem, an algorithm must determine which agent should move to which goal. Such an assignment between starts and goals is called a matching. This problem is therefore named multi-agent pathfinding with matching (here abbreviated to *MAPFM*).

In this paper, first *MAPFM* will be defined, and after that possibilities will be discussed to extend *M** to solve *MAPFM* problems as well. To do this, two methods will be proposed. These two methods will be compared, both to each other, and to a number of other algorithms solving *MAPFM*. Apart from this comparison, a number of extensions to *M** will be considered which improve *M**'s performance. Some of these extensions can also improve the performance of *M** on plain *MAPF* problems.

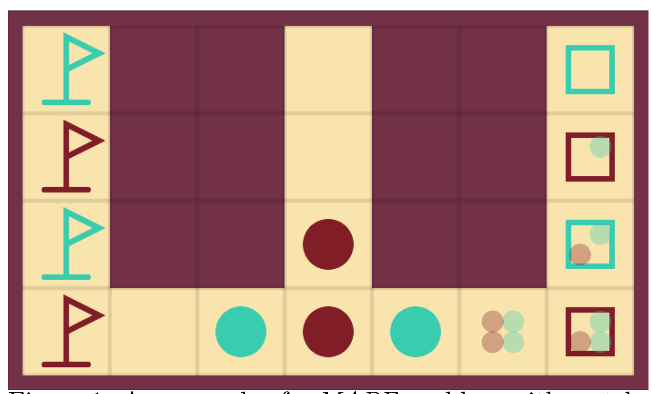


Figure 1: An example of a *MAPF* problem with matchings. Two teams (of different colours) of agents (circles) moving from their starts (squares) to their goals (flags).

1. Prior work

Before this research, a separate study has been performed [7] on a problem called target-assignment and path-finding (*TAPF*). The difference between *TAPF* and *MAPFM* is that in *TAPF*, the *makespan* (the cost of the longest path) instead of the *sum of individual costs* (the cost of all paths combined) is minimised. To solve *TAPF*, [7] uses conflict based search, with a max-flow based algorithm to solve matchings within one team. It is yet unclear if it is possible to use such a max-flow based algorithm for *MAPFM*. However this is not a question that will be answered in this paper. Parallel research to this will however attempt to find an answer to this question [8].

Solving *TAPF* and *MAPFM* with other algorithms than *CBS* has not yet been explored.

In contrast, a lot of research has been done finding and improving algorithms which solve *MAPF* [5, 6, 9–13]. But which of these techniques can be efficiently extended to also solve *MAPFM* is still an open question. This paper attempts to answer this question specifically for *M**.

2. Definition of *MAPFM*

The definition of multi-agent pathfinding used in this paper is based on Stern's definition given in [14]. The definition is as follows:

A *MAPF* problem consists of the following elements:

$$\langle G, s, g \rangle$$

- G is an undirected graph $\langle V, E \rangle$
  - V is a set of vertices
  - E is a set of edges between vertices
- s is a list of k vertices where every s_i is a starting position for an agent a_i
- g is a list of k vertices where every g_i is a target position for an agent a_i

Although the algorithms presented in this paper would work on any graph G, in examples and experiments, G is simplified to a 4-connected grid where edges have a weight of 1. In this paper, this grid is sometimes called a 'map', on which agents move from their starts to their goals.

This definition is now expanded to a definition of multi-agent pathfinding with matching, called *MAPFM*. The definition of this new problem is as follows:

$$\langle G, s, g, sc, gc \rangle$$

- sc is an array of colours sc_i for each starting vertex s_i
- gc is an array of colours gc_i for each target vertex g_i

This definition divides all agents into teams. An agent a_i's team colour is the colour of its starting location sc_i. In total there are K teams $k_1 \dots k_n$. K is equal to the number of different colours in sc and gc.

In *MAPFM*, a goal state is a state in which each agent is on a goal location in g, such that the colour of every agent is equal to the colour of the goal they are on.

MAPFM disallows vertex and edge conflicts as described in [14]. This means that two agents a_i and a_j cannot be at the same vertex v or edge e at a timestep t.

MAPFM is an *optimisation problem*. A solution to this problem has a cost c which needs to be minimised. The cost of an agent a_i's path is defined to be the number of timesteps it takes for an agent to reach its goal and never leave it again. This means that it's possible for an agent to reach the goal in an earlier timestep, and leave it again later to let another agent pass.

To find the cost of a solution for all agents, [14] presents two methods. In this paper, the *sum of individual costs* is minimised. The cost of a solution is thus the sum of the cost of all agents their paths.

3. A description of *M**

*M** [5] is a pathfinding algorithms which searches a search space, similar to *A**, but is specifically designed to work with multiple agents. In *M**, the search space consists of the positions of all the agents on a grid. To use *A** for multiple agents, the planning of paths for all agents needs to be coupled to make sure no states are expanded in which collisions occur.

Unlike *A**, in *M** the planning of agents is initially not coupled. Instead, *M** assumes that the optimal path for an agent to their goal does not contain any collisions. As long as this is true, the planning of agents is separated.

However, the assumption that all optimal paths are collisionless obviously does not always hold. Therefore, each state in *M** also holds, apart from the positions of each agent, two sets called the *collision set* and the *backpropagation set* as described in [5]. When *M** detects that a state contains collisions, the algorithm does not continue expanding this state. It instead uses information stored in these two sets to backtrack and find the shortest route around these collisions. Agents associated in the collision temporarily plan routes in a coupled fashion.

After collisions, *M** plans agents independently again according to their individually optimal path. It does, however, record information about the previous collisions (in these *backtracking*- and *collision sets*). This is to make sure that when in the future another collision occurs, it can either resolve this locally or backtrack back to the previous collision to plan it differently to potentially avoid this new collision altogether.

[5] proves that *M** provides optimal solutions to *MAPF*.

4. *M** and matching

To add matching to *M**, this paper proposes two options which are proposed to be named "inmatching" and "prematching". In this section, both are explained and their advantages and disadvantages are discussed.

4.1. Inmatching

Inmatching is the process of performing matching as a part of the pathfinding alorithm that is used. To understand it, it is useful to first look at inmatching in *A**. With *A**, the expansion of a state consists of all possible moves for all agents. *A** searches through the search space, until the goal state has been removed from the frontier. An admissible heuristic *A** will guarantee that following the children of this first goal state gives a shortest path between the start and the goal.

With *inmatching*, there is not one goal state. Instead, any state in which all agents stand on a goal of their own colour is considered a goal state. This means there is more than one goal state. To keep the heuristic admissible, the distance to the nearest goal state is used as a heuristic.

Inmatching applied to *M**

Inmatching can similarly be used with *M**. However, there is something that makes it much less efficient in *M** than in *A**, the implications of which will be evaluated in section 7.1.

One of the core ideas of *M** is that agents, when not colliding, follow an individually optimal path to their goal. With *inmatching*, every goal in the team of the agent is a valid goal. Therefore, to preserve optimality, agents should consider optimal paths to each of the goals in the team.

To understand the implication of this, a concept needs to be explained first: the *expansion size*. In *A**, a priority queue is used with states in it. Each time a state is removed from the queue it is expanded into a number of child states which go back into the queue. The number of child states which are expanded is the *expansion size*.

In regular *M**, when there are no collisions, the *expansion size* is 1. Each agent simply follows their individually optimal path to their goal. Only when there are collisions, the *expansion size* is bigger than 1 which is less efficient. But with *inmatching*, there is not one optimal path. Thus, the expansion size is often bigger than 1 even when there are no collisions. In fact, the expansion size grows exponentially with then number of goals in a team. The following equation gives an upper bound for this expansion size:

$$\prod_{n=1}^{K} \text{number of goals of team}(k_n)$$

*M** using *inmatching* will from now on be abbreviated with *imM**

4.2. Prematching

Alternatively, there is *prematching*. With *prematching* the *MAPFM* problem is transformed in a number of *MAPF* problems. Each possible matching is calculated in advance, and normal *M** as described by [5] is performed on each matching. Algorithm 1 shows how this is done.

*M** using *prematching* will from now on be abbreviated with *pmM**

Algorithm 1: prematch *M**

Result: Find the matching with smallest cost

---

foreach $m \leftarrow$ *find all matchings*$(starts, goals)$ **do**
  $S(i) \leftarrow \text{mstar}(starts, m)$ {Evaluate with *M**}
end
return *min(S) {by calculated cost}*

5. Extensions to *M**

*M** can be improved upon in various ways. Some of these extensions are only applicable to matching *M**, while others improve *M** itself.

In this section these extensions are explained and their effects are presented.

5.1. Precomputed paths and heuristics

A part of solving *M** is finding the individually optimal path for each agent. This can be done with conventional pathfinding algorithms such as *A**. But to avoid repeated recalculation, it is important to first create a lookup table of the distance to each goal, for every open square on the map.

When using *prematching*, the same goals and starts are (although in a different permutation) reused multiple times. The lookup table can remain the same every time.

Using this lookup table, the heuristic *M** can be improved as well. Instead of using the *euclidean-* or *Manhattan distance* to the goal, the lookup table can be used to find the exact distance. The heuristic found with this approach also takes the obstacles on the map into account.

5.2. Operator decomposition

In [6], Standley describes a technique called operator decomposition (*OD*). With operator decomposition the number of expanded nodes is divided by n (where n is the number of agents), while the search depth is multiplied by n.

To do this, expansions can be partial or complete. Each time a state is expanded, only the moves of one agent are expanded and put back in the queue. Only when this partially expanded node comes to the top of the queue, the moves of another agent are considered. Once all agents in a state have expanded, the state is said to be complete again.

The queue now contains both partial and complete states. This provides the search algorithm with more granularity in prioritising moves, which in turn can mean a considerable improvement in runtime.

M* can also benefit from operator decomposition as explained in [15].

5.3. Pruning of matchings

*pmM**, as previously described, usually has to evaluate every matching of every team. However, it is possible to discard some matchings without evaluating them with *M** at all, by using heuristics.

To do this, first calculate the heuristic (i.e. the sum of distances between starts and goals) of the initial state of every matching. Because this heuristic is admissible, it represents a lower bound for the cost of this matching.

Now, before every evaluation of a matching m, its heuristic is first checked against the best matching m_{best} found so far. If heuristic$(m) \geq cost(m_{best})$, there is no need to evaluate m and m can be pruned.

5.3.1. Sorting

To take maximum advantage of pruning, matchings can be sorted based on their heuristic. Making sure the matching with the lowest heuristic is evaluated first can increase chances that later matchings are pruned. Algorithm 2 shows how pruning with sorting works.

Algorithm 2: prematch *M** with pruning

```
Result: Find the matching with smallest cost
        without evaluating every matching.
m_best ← ∅
──────────────────────────────
M ← find all matchings(starts, goals);
sort(M) {on heuristic; ascending}
foreach m ← M do
    if heuristic(m) < cost(m_best) then
        s ← mstar(starts, m);
        m_best ← min(m_best, s);
    end
end
```

6. Experimental setup

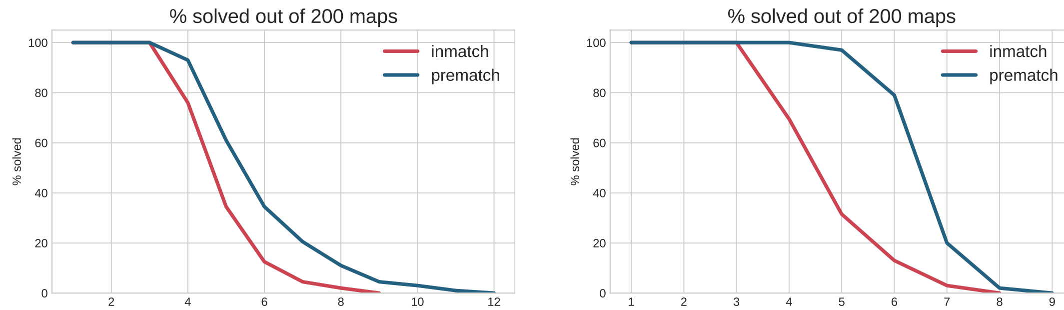
To evaluate the performance of matching *M**, a number of experiments were performed. For each of these experiments, the algorithm used was written in **python 3.9** and benchmarks were run on a virtualized system with a 12 core 12 thread Intel Xeon E5-2683 running at 2GHz, which has 8Gb of RAM.

There are various factors deciding how an algorithm or variant of an algorithm performs in benchmarks. Three important factors are the total number of agents for which paths need to be found, the number of teams over which the agents are distributed, and the layout of the maps on which benchmarks are run.

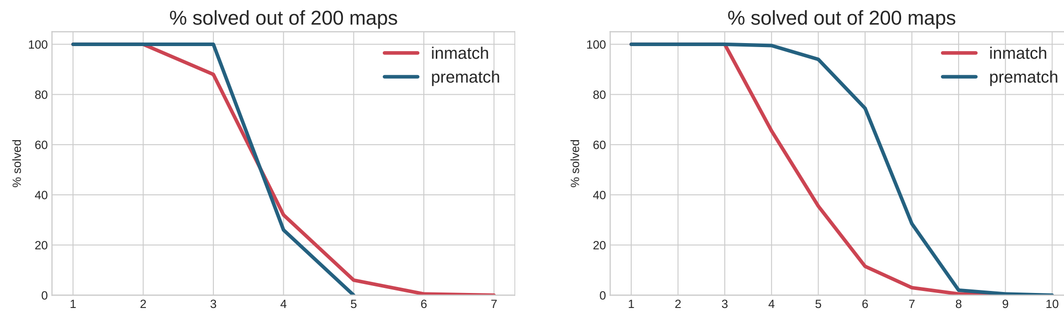
To show the differences in performance as these parameters vary, all experimental are grouped into sets of four *scenarios*. Performance is assessed on maps where either 25% or 75% of the grid is an obstacle, and in situations where agents are grouped into either 1 or 3 teams.

For these benchmarks, maps are randomly generated. Four sets of maps were generated for each experiment with the previously discussed parameters. Agents and goals are uniformly distributed on this map. Only maps in which each individual agent is capable of reaching their goal are used to greatly reduce the chance that maps are impossible to solve.

Because *MAPFM* is an *NP-Hard* problem, it is possible that finding the solution to an instance of the problem takes an unreasonable amount of time. Therefore, in each experiment, a timeout of 2 minutes is used. Experimental results do not show the time it takes for an algorithm to find a solution (the reason for which will be explained in section **??**). Instead they show the percentage of maps (out of a set of 200) which the the algorithm manages to solve within this timeout of 2 minutes.



(a) Agents split over 3 teams. Left: 75% wall, right: 25%wall



(b) All agents in 1 team. Left: 75% wall, right: 25%wall

Figure 2: Percentage of maps solved in 2 minutes out of 200 random 20x20 maps. In graphs showing times to solution, the line represents the 50th percentile, while the shaded area around goes up to the 90th percentile and down to the 10th percentile.

7. Results and Discussion

7.1. Matching strategies

In section 4, two strategies were proposed to add matching to *M**. Both strategies were tested following the experimental setup described in section 6. In figure 2 the results of this experiment are shown.

The figures show that *prematching* is generally superior to *inmatching*. In all but one of the *scenarios*, *prematching* performs better. In section , it was hypothesised that this could happen, because of the larger expansion size.

The reason for this difference is the same as was hypothesised in section . *inmatching* leads to a large expansion size.

Expansion size

To prove this was indeed true, another, separate, experiment was performed. Again on 200 random maps as described in section 6, but this time showing the average number of states expanded each expansion.

In 3, on a logarithmic scale, the outcomes of this experiment are shown. Indeed, on average, inmatching expands many more states.

When *inmatching* is better

However, in one of the scenarios in figure 2b *inmatching* does seem to perform slightly better than *prematching*. In this benchmark, all agents are in a single team, and the map is 75% filled with obstacles.

When agents are all in one team, there are more matchings than when agents are separated into multiple teams. With *prematching*, each matching needs to be exhaustively searched which is slow.

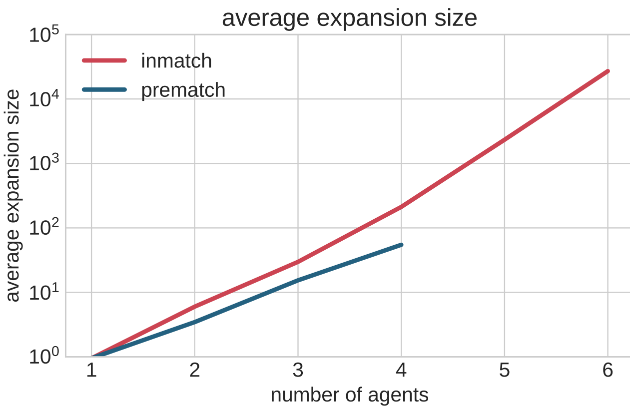
In contrast, the problem of *inmatching* expanding into many states is reduced in this scenario. There are fewer directions for agents to go because of the large number of obstacles.

So in general, when there are many matchings, but few options for agents to move in, *inmatching* can perform better.

7.2. Extensions to *M**

Various extensions have been proposed in section 5. To show the benefits of each extension, they were compared following the experimental setup described in section 6.

In the graphs of figure 4 are shown. Each line in the graph shows a new extension added to the algorithm,



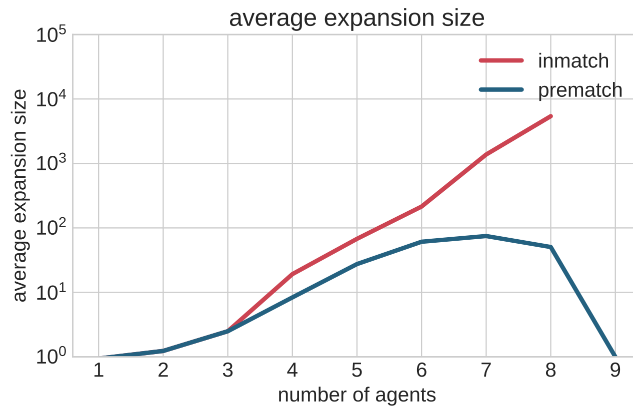


Figure 3: Average expansion size on the same maps as used in figure 2b (ie. all agents in one team). Left: 75% wall, right: 25%wall.

ordered following the legend.

Because in section 7.1, *prematching* was observed to generally perform better, only the *prematching* matching strategy is used in this experiment.

Pruning and sorting

In the graphs where 25% of maps are obstacles, pruning and sorting have a large impact on performance. However, in graphs where 75% is a wall there is barely a difference at all. The heuristic used to prune, is taking the Manhattan distance between the agent start locations and the closest goal. The heuristic used here is taking the Manhattan distance from an agent to the closest goal. This does not take into account the obstacles in the map.

Pruning is only possible when the heuristic is larger than the best matching found so far (as described in section 5.3). This does not happen often with this inaccurate heuristic. Therefore, using the precomputed heuristic (also shown in figure 4) makes such a large difference.

When there are less obstacles in the map, the probability of pruning is much higher explaining the difference in performance of pruning and sorting between these two scenarios.

Memory usage

The best variant of *M** can be seen solving maps with 14 agents in figure 4a. After this, the percentage abruptly drops to 0. This is not caused by the complexity of the problems, but rather by the memory that *M** uses here.

Attempts with maps with 15 agents saw memory usages of over 8GiB to solve a single instance due to which attempts were stopped by the operating system.

7.3. Comparison with other algorithms

This research is part of a set of parallel studies on how to extend a collection of *MAPF* algorithms to give them the ability to solve matchings. These are

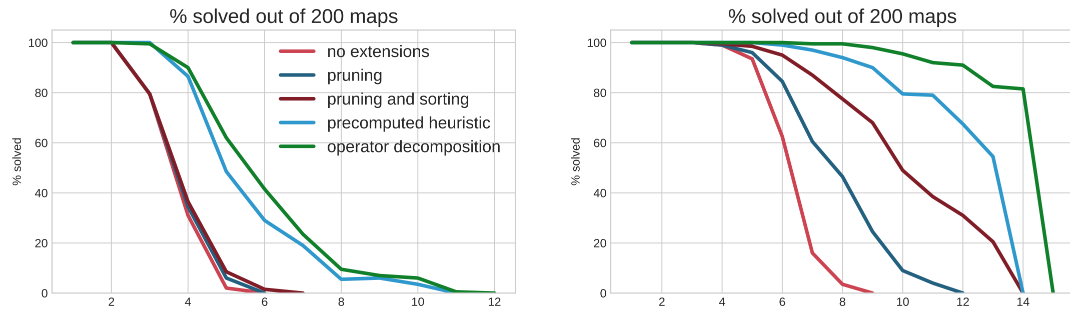
- Extended partial expansion *A** (*EPA**) [10] (implementation and research by [16]).
- *A** with operator decomposition and independence detection (*A*+OD+ID*) [6] (implementation and research by [17])
- Increasing cost tree search (ICTS) [13] (implementation and research by [18])
- Conflict-Based Min-Cost-Flow (CBM) [7] (implementation and research by [8])

Experiments were performed following the experimental setup described in section 6. All algorithms were benchmarked on the same computer to ensure a fair comparison.

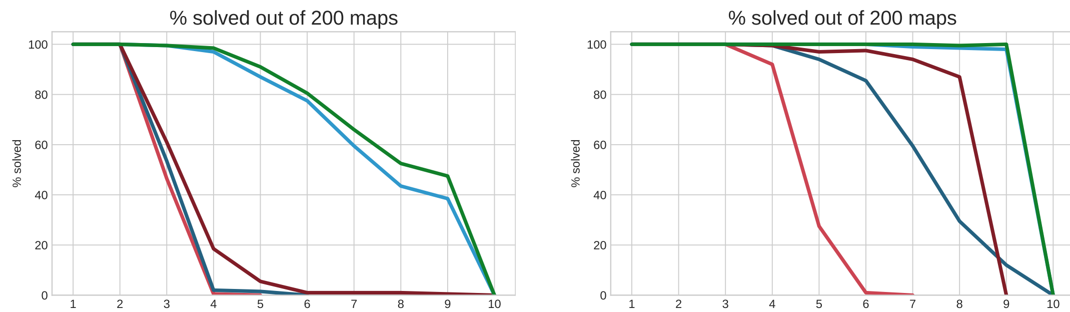
8. Future work

Partial Expansion

One problem with the *inmatching* strategy presented in section 7.1 is that the number of states which expanded from a single state is very large. However, a lot of the expanded states are never needed or accessed again. A system such as *partial-* or *enhanced partial expansion* as presented in [10] and [11] might allow *inmatching* to compete with *prematching*. But even for *prematching* it might make a difference when there are many agents. While creating the graphs in figure 4, the data points where 14 or 15 agents were involved could use as much as 8GiB of memory.



(a) 3 teams. Left: 75% wall, right: 25%wall



(b) 1 team. Left: 75% wall, right: 25%wall

Figure 4: Percentage of maps solved in 2 minutes out of 200 random 20x20 maps. Each line and extension to M^* is graphed in combination with all previous extensions. For example, the line displaying *operator composition* also uses precomputed heuristics and all other previous extensions.

Conflict based search

In this paper, matching M^* was compared to several other algorithms. One of those was CBS [12]. However, this implementation used a max-flow based approach similar to the one described in [7], but altered to find the *sum of individual costs* instead of the *makespan*.

CBS, without matching and these max flow extensions, is one of the most studied algorithms for *MAPF* ([12, 19–22]). Comparing M^* with *inmatching* and *prematching* to *CBS* using similar approaches to add matching to it, could provide useful data.

In addition to that, it may be interesting to combine M^* and *CBS*. Normally, the low level search of *CBS* is an A^* based solver for each agent. When adding matching using max flow, in the low level an entire team is planned using max flow. Instead it may be possible and beneficial to substitute this max flow solver with some version of M^* to plan a team.

Waypoints

In previous research [23–26], *MAPF* was extended with waypoints instead of matching. It may be possible to combine these two extensions. For example, each team may have a set of waypoints. It doesn't matter which agent visits which waypoint as long as all waypoints are visited. Or alternatively, each agent has their own waypoints, but the goals are still shared with a team of other agents.

Agents and goals

In the definition of matching presented in this paper, in each team the number of goals was always equal to the number of agents. However, in real world scenarios this may not always be the case.

For example, there may be a number of robots who have a number of tasks to do. Some robots cannot perform certain tasks so robots are teamed. But there may be more tasks than robots in each team, and robots will need to prioritise.

Some research has already been done in this area, [27], but this uses *TAPF* as a basis.

9. Reproducibility

Results in this paper have been generated using an implementation of M^* made in python specifically for this research. The code for this is publicly available on github at https://github.com/jonay2000/research-project doubly licensed under the Apache and MIT li-

(a) 3 teams. Left: 75% wall, right: 25%wall

(b) 1 team. Left: 75% wall, right: 25%wall

Figure 5: TODO

censes. Reports of bugs and new additions to this codebase are always welcomed.

10. Conclusion

In this paper, a new addition to multi agent pathfinding (*MAPF*) was introduced, called matching. This new problem, is called *MAPFM M**, an algorithm for *MAPF*, was modified to solve these *MAPFM* problems. To do this, two strategies were explored called *inmatching* and *prematchings*. Experimental results showed that in many cases, *prematching* is superior to *inmatching*.

After that, several new and existing improvements to *prematching M** were considered and their benefits were experimentally evaluated.

Finally, it was shown ... how *prematching M** compares to other algorithms (TODO, say that it does well, or poorly compared to them depending on results).

References

[1] J. Mulderij, B. Huisman, D. Tönissen, K. van der Linden, and M. de Weerdt, "Train unit shunting and servicing: A real-life application of multi-agent path finding," 2020.

[2] J. Li, A. Tinka, S. Kiesel, J. W. Durham, T. Kumar, and S. Koenig, "Lifelong multi-agent path finding in large-scale warehouses," 2020.

[3] A. Mahdavi and M. Carvalho, "Distributed coordination of autonomous guided vehicles in multi-agent systems with shared resources," in *2019 SoutheastCon*, IEEE, 2019, pp. 1–7.

[4] J. E. Hopcroft, J. T. Schwartz, and M. Sharir, "On the complexity of motion planning for multiple independent objects; pspace-hardness of the" warehouseman's problem"," *The International Journal of Robotics Research*, vol. 3, no. 4, pp. 76–88, 1984.

[5] G. Wagner and H. Choset, "M*: A complete multirobot path planning algorithm with performance bounds," in *international conference on intelligent robots and systems*, IEEE, 2011, pp. 3260–3267.

[6] T. Standley, "Finding optimal solutions to cooperative pathfinding problems," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 24, 2010.

[7] H. Ma and S. Koenig, "Optimal target assignment and path finding for teams of agents," 2016.

[8] R. Baauw, "Todo," 2021.

[9] E. Lam, P. Le Bodic, D. D. Harabor, and P. J. Stuckey, "Branch-and-cut-and-price for multi-agent pathfinding.," in *IJCAI*, 2019, pp. 1289–1296.

[10] M. Goldenberg, A. Felner, R. Stern, G. Sharon, N. Sturtevant, R. C. Holte, and J. Schaeffer, "Enhanced partial expansion a*," *Journal of Artificial Intelligence Research*, vol. 50, pp. 141–187, 2014.

[11] A. Felner, M. Goldenberg, G. Sharon, R. Stern, T. Beja, N. Sturtevant, J. Schaeffer, and R. C. Holte, "Partial-expansion a* with selective node generation," *Proceedings of the 5th Annual Symposium on Combinatorial Search, SoCS 2012*, pp. 180–181, 2012.

[12] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2015.

[13] G. Sharon, R. Stern, M. Goldenberg, and A. Felner, "The increasing cost tree search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 195, pp. 470–495, 2013.

[14] R. Stern, N. Sturtevant, A. Felner, S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T. Kumar, *et al.*, "Multi-agent pathfinding: Definitions, variants, and benchmarks," 2019.

[15] C. Ferner, G. Wagner, and H. Choset, "Odrm* optimal multirobot path planning in low dimensional search spaces," in *2013 IEEE International Conference on Robotics and Automation*, IEEE, 2013, pp. 3854–3859.

[16] J. de Jong, "Todo," 2021.

[17] I. de Bruin, "Todo," 2021.

[18] T. van der Woude, "Todo," 2021.

[19] E. Boyarski, A. Felner, R. Stern, G. Sharon, D. Tolpin, O. Betzalel, and E. Shimony, "Icbs: Improved conflict-based search algorithm for multi-agent pathfinding," in *Twenty-fourth international joint conference on artificial intelligence*, 2015.

[20] A. Felner, J. Li, E. Boyarski, H. Ma, L. Cohen, T. S. Kumar, and S. Koenig, "Adding heuristics to conflict-based search for multi-agent path finding," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 28, 2018.

[21] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Meta-agent conflict-based search for optimal multi-agent path finding.," *SoCS*, vol. 1, pp. 39–40, 2012.

[22] J. Li, A. Felner, E. Boyarski, H. Ma, and S. Koenig, "Improved heuristics for multi-agent path finding with conflict-based search.," in *IJCAI*, 2019, pp. 442–449.

[23] N. Jadoenathmisier, "Extending cbs to efficiently solve mapfw," 2020.

[24] A. Michels, "Multi-agent pathfinding with waypoints using branch-price-and-cut," 2020.

[25] S. Siekman, "Extending a* to solve multi-agent pathfinding problems with waypoints," 2020.

[26] J. van Dijk, "Solving the multi-agent path finding with waypoints problem using subdimensional expansion," 2020.

[27] V. Nguyen, P. Obermeier, T. C. Son, T. Schaub, and W. Yeoh, "Generalized target assignment and path finding using answer set programming," in *Twelfth Annual Symposium on Combinatorial Search*, 2019.